Event-driven Approach for Logic-based Complex Event Processing

Darko Anicic[1] darko.anicic@fzi.de
Paul Fodor[2] pfodor@cs.sunysb.edu
Roland Stühmer[1] roland.stuehmer@fzi.de
Nenad Stojanovic[1] nenad.stojanovic@fzi.de

[1]*FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany*
[2]*Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, U.S.A*

***Abstract*—In this paper, we present a powerful logical encoding of complex event patterns into Transaction Logic programs. Complex Event Processing (CEP) deals with finding composed events and has useful applications in areas ranging from agile business and enterprise processes management, financial market applications to active Web and service oriented computation. Many systems for event processing have ad-hoc semantics with unexpected behaviors. Hence formal logical semantics is an important requirement for event-driven reactive systems. On the other hand, many logic-based approaches for CEP (based on formal semantics) fail, due to their inability to compute complex events in the data-driven fashion. Our approach enables both logic-based and data-driven complex event detection. Moreover, the proposed backward chaining approach allows for very efficient reasoning of complex events and actions triggered by these events.**

***Keywords*-Complex Event Processing; Event-driven backward chaining rules;**

I. Introduction

Complex Event Processing (CEP) has the task of processing multiple events with the goal of identifying meaningful event patterns. Detected event patterns are then used to trigger response actions, which in turn may trigger other events. In this cycle, a reactive system is supposed to achieve some useful job. Completeness of the job can be identified either with a particular state being reached (for finite reactive systems), or a set of properties (constraints) being satisfied all the time during the execution (for infinite reactive systems). In both cases (including a mix of them) we have a particular goal or goals that need to be fulfilled. It has been recognized elsewhere [1], [2], [3], [4] that some sort of logic is required to keep event-driven systems running in a controlled manner. The logic should ensure correctness of an event-driven execution. It means that the execution is handled in a way which guarantees satisfiability of predefined constraints on states (in a state-changing environment). Further on, the logic can serve to detect a desired state that a user might be interested in; to trace the state of any entity, at any time; to discover conflicting states etc. However declarative approaches also have their own limitations. The main common drawback is their inability to do an effective *event or data-driven* computation of event patterns. Data-driven computation means that an event pattern is detected as soon as the last event required for a complete match of the pattern has occurred. This way of fulfilling goals (i.e., detecting event patterns) is also called *forward chaining*. On the other hand, logic-based approaches are *query or goal-driven*. This means that, for a given event pattern, an event processing engine needs to check if this pattern has been satisfied or not. The check is performed at the time when such a query (goal) is set. Therefore, this way of computation is called query or goal-driven (also known as *backward chaining*). If satisfied by the time when it is checked, the pattern will be triggered. If not, the pattern cannot be detected till the next time the same check is performed (though it can become satisfied in between the two checks). This happens as the goal-driven systems try to prove a path *backwards* from a given goal (i.e., event pattern) to the current state (rather than trying to find a path *forward* from the current state that achieves that goal). Backward approaches attempt to *deduce* an event pattern rater than to *detect* it (as soon as it happens). This is why they have typically been suitable for *request-response* systems. These systems detect an event based on a request, followed by an action which is a response. Further on, backward approaches are suitable for periodic detection of event patterns (e.g., each day, every month etc. over historic data). However, they fail when patterns need to be detected as soon as they really occur.

In this paper we present an approach which is based on a *logic*, and still it is *event or data-driven*. This approach specifies complex event patterns declaratively, which allows for powerful *logical inference*. Since the notions of event patterns, states and response actions are all put in a unified logical framework, the inference process (specifically deduction) can be utilized over all of them. For example, a set of state-changing actions can be triggered by detected event patterns. During the execution they may produce other events, which will be used for detection of other event patterns and so on. The new patterns can further be used to synchronize the ongoing actions; control their execution w.r.t order in which they are executed; trace the state of any of these actions such that certain constraints are met at any time (otherwise the current state will be roll-backed to the last one which meets the constrains), etc.

Our approach is based on decomposition of complex event patterns into *intermediate patterns* (i.e., *goals*). The status

of achieved goals at the current state shows the progress toward matching of one or more event patterns. Goals are automatically asserted as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or complete pattern. Important characteristics of these goals are that they are asserted only if they are used later on (to support a more complex goal or an event pattern), goals are all unique, and goals persist as long as they remain relevant (after the relevant period they are deleted). Goals are asserted by declarative rules, which are executed in the backward chaining mode. A specific property of these rules is that they are event-driven.

The paper is organized in the following manner. Section II reviews related work, and compares it to our work. Section III introduces Concurrent Transaction Logic ($\mathcal{CTR}$) as the main underlying formalism of our logic-based approach. In Section IV we define an event pattern, and explain the basic concepts of event processing based on that definition. Section V describes in details data-driven event detection, and implements all event operators defined in Section IV. Finally, in Section VII we summarize our work and give an outline for the future work.

II. Related Work

In order to capture relevant changes in a system and respond to those changes adequately, a number of logic-based reactive frameworks have been proposed. This section briefly overviews some approaches for events and reactive rules processing. Work on modeling *behavioral* aspects of an application (using various forms of reactive rules) started in the Active Database community a long time ago. Different aspects have been studied extensively, ranging from modeling and execution of rules, to discussing architectural issues [5]. However what is clearly missing in this work is a clean integration of active behavior with *deductive* and *temporal* capabilities. This is exactly a goal of our approach. Work in [1] goes toward this direction, i.e., putting event patterns in a *logical framework*. As pointed out there, rules can be effectively used for describing, so called, "virtual" event patterns. There exist a number of other reasons to use rules, i.e., rules serve as an abstraction mechanism and offer a higher-level event description. Rules allow for an easy extraction of different views of the same reactive system. Rules are suitable to mediate between the same events differently represented in various interacting reactive systems. Finally, rules can be used for reasoning about causal relationship between events. However in [1], the authors focus only *event queries*, and not on event patterns detectable in a *data-driven* fashion. Discovery of complex events by posing event queries is by no means useless (e.g., querying past data, such as log files to detect new events that happened). However it is not sufficient when we talk about CEP. CEP aims to detect complex patterns in *real-time*, i.e., as soon as they occur (and not at the moment when a reactive system poses a query).

In [2], a homogeneous reaction rule language was proposed. From a linguistic standpoint, the approach combines different paradigms such as complex events, reactive rules, declarative rules and integrity constraints. However it suffers from the same drawbacks as mentioned in [1], i.e., complex events are discovered at the moment when corresponding queries are processed, rather than when they have really happened. For instance, the language enables for construction of queries that are trying to discover new events periodically (e.g., every 10 seconds). Clearly such an approach raises issues about performance and practical usability of the language despite its numerous strengths.

The problem with these and other logic-based approaches is that event processing relies on inference engines with no capability to do event-driven processing. Inference procedures are either based on backward or forward chaining of rules. As mentioned above, backward chaining (or goal-driven reasoning) may be employed to find suitable data (i.e., simple events) answering the question of validity of a given rule (i.e., event pattern). This process is initiated in a top-down manner, starting from a given rule, down to to the simple facts. Forward chaining starts bottom-up with facts available from the database, and aim to deduct *all possible* truths grounded by a set of rules. Our work contrast afore mentioned approaches by implementing powerful event processing in a *logical manner*, yet enabling their *data-driven* computation (and focused only on derivation of required goals).

The closest to our approach is [3], an attempt to implement business rules with Rete Algorithm using subgoals. The work deals with data-driven backward chaining. It has deductive capabilities, and detect fulfilled conditions in business rules (using backward chaining), as soon as relevant facts become available. In our work, we focus rather on complex event detection, and can unify in a logical framework not only events but conditions and actions (i.e., reactions on events) too.

[4] is an attempt to combine ECA rules with Process Algebra. The idea is to enrich the *action* part, with the declarative semantics of Process Algebra, particularly CCS algebra [6]. Use of Process Algebra specification aims to enable the *reasoning* capability in such an ECA system. Once again, our work differentiates from [4] as it integrates powerful complex event processing with rich conditional capabilities and actions, all in one declarative framework.

Concluding this section, all mentioned studies are motivated to use more formal semantics. Our approach based on $\mathcal{CTR}$ may also be seen as an attempt towards that goal, though followed by a pure Logic Programming style. Apart from this, it feature data-driven computation of complex events as well as reach deductive capabilities.

III. CTR Overview

Concurrent Transaction Logic ($\mathcal{CTR}$) [7] is a general logic designed specifically for the rule-based paradigm, and intended to provide a declarative account for state-changing actions. A model and proof theory of $\mathcal{CTR}$ has been defined in [7]. It is an extension of Transaction Logic ($\mathcal{TR}$) [8], with its "Horn" fragment defined. In this section we will informally introduced syntax and semantics of $\mathcal{CTR}$. For further considerations related to the semantics of $\mathcal{CTR}$ the reader is referred to [8].

The atomic formulas of $\mathcal{CTR}$ has the form $p(t_1, ..., t_n)$, where p is a predicate symbol and the t_i's are function terms. The elementary transitions **ins/1** and **del/1** specify basic updates of the current state of the database executed by an oracle that takes the database into a new state by inserting or deleting an atom in the extended database. Complex formulas are built out of atomic formulas and elementary transitions using connectives and quantifiers.

The same as in classical logic, $\mathcal{CTR}$ has $\wedge, \vee, \neg, \forall$, and $\exists$. Unlike in classical logic, $\mathcal{CTR}$ has two connectives, $\otimes$ (serial conjunction) and $|$ (concurrent conjunction); and $\odot$ (modality of isolation, i.e., isolated execution).

In short, $\mathcal{CTR}$ is a logic for state-changing actions. The truth of $\mathcal{CTR}$ formulas is determined over *paths*. A path is a finite sequence of states. If a formula, ψ, is true over a path $< s_1, ..., s_n >$ it means that ψ can be executed starting with state s_1. During the execution, ψ will change the current state to $s_2, s_3, ...$ and finally terminate at the state s_n. Having this in mind, the intended semantics of $\mathcal{CTR}$ connectives and modal operators can be summarized as follows:

- $\phi \otimes \psi$ means: execute ϕ, then execute ψ;
- $\phi \mid \psi$ means: execute ϕ and ψ concurrently;
- $\phi \wedge \psi$ means: ϕ and ψ must both be executed along the same path;
- $\phi \vee \psi$ means: execute ϕ or ψ nondeterministically;
- $\neg\, \phi$ means: execute in any way, provided that this will not be a valid execution of ϕ;
- $\odot\ \phi$ means: execute ϕ in isolation of other possible concurrently running activities;

The logic has notions of *data* and *transition* oracles. The *data* oracle, $\mathcal{O}^d(\mathbf{D})$, is used to solve queries related to a particular state $\mathbf{D}$. Likewise, the *transition* oracle, $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, is used to specify an update transition: **ins/1** and **del/1**. If $a \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, then a is an elementary update that changes state $\mathbf{D}_1$ into state $\mathbf{D}_2$. For example, the state change can happen if an atom or a Horn rule is inserted or deleted from $\mathbf{D}$. For further details about $\mathcal{CTR}$ semantics, its "Horn" fragment etc., the reader is referred to [7], [8], [9].

IV. Putting Events in a Logical Framework

The execution in CEP applications is driven by events. In order to react on an event, we need first to specify event patterns which we want our application to react on. An *event pattern* is a template which matches certain events. For example, an event pattern matches all orders from customers in response to a discount announcement event. Sometimes the discount announcement event is called an *atomic* event, which is used to build a *complex* event. In general, an event pattern is a pattern which is built out of (atomic and complex) events satisfying certain event operators and/or time constraints. Event operators typically depend on a particular language used for CEP. For example they can be a conjunction, disjunction, causality dependencies, or negation of events etc. Time constraints specify different time conditions that must be satisfied by a complex event (e.g., one event happened 10 min after another one, or they both must happen within a certain time interval).

We assume a discrete time model, where ***time*** is an ordered set of time points. In this paper points are represented as integers, but other time models for time and data representation are possible without restrictions. Since $\mathcal{CTR}$ is a state-changing logic (see Section CTR Overview), the notion of an event pattern is defined as a *relevant state change* in an event-driven system, characterized by the time. Formally, an event is $e^{[T_1,T_2]}(X_1, X_2, ..., X_n), n \geq 0$, where e is name of an event pattern (i.e., a predicate symbol), and $X_1, X_2, ..., X_n$ is a list of arguments, representing data parameters (terms). Event patterns contain data relevant for a reactive system. The data of an event pattern is a data term that may be either a variable, a constant, or a function symbol. $[T_1, T_2]$ defines a time interval during which the event has occurred. Following the argumentation from [2] (to prevent unintended semantics in event processing) events are defined on intervals (i.e., $[T_i, T_j]$), rather than on event points (i.e., T_i).

Definition IV (Event Pattern). An event pattern is a formula of the following form:

- an atomic event;
- $(event_1 \otimes event_2 \otimes ... \otimes event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Sequence of events);
- $(event_1 \wedge event_2 \wedge ... \wedge event_n)$, where $n \geq 0$ and each $event_i$ is an event (Classical conjunction);
- $(event_1 | event_2 | ... | event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Concurrent conjunction);
- $(event_1 \vee event_2 \vee ... \vee event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Disjunction);
- $\neg event$, where $event$ is an event pattern (Negation).

A rule is a formula of the form $eventA \leftarrow eventB$, where $eventA$ is an atomic event, and $eventB$ is either an atomic or a complex event pattern ■

In the above definition, every $event_i$ is defined over a time interval $[T_1, T_2]$ with possible list of data terms that are omitted due to space reasons.

In following examples we demonstrate the power of event patterns defined using syntax of $\mathcal{CTR}$, and give justification for their use in practice.

Example 4.1 defines a complex event, $checkStatus$, which happens "*if a priceChange event is followed with a stockBuy event*". Further on, the two events have happened within a certain time frame (i.e., $t < 5$).

Example 4.1.

$checkStatus^{[T_1,T_4]}(X,Y,Z,W) \leftarrow priceChange^{[T_1,T_2]}(X,Y) \otimes stockBuy^{[T_3,T_4]}(Z,Y,W) \otimes (T_4 - T_1 < 5)$.

assuming we have defined the following event patterns:

- $priceChange^{[T_i,T_j]}(X,Y)$ is an event pattern, that describes the change in the stock price X (e.g., $\pm 5\%$) of a company Y;
- $stockBuy^{[T_i,T_j]}(Z,W,Y)$ defines a transaction, in which, a buyer Z has bought W amount of stocks from a company Y.

In some cases a user may be interested in analyzing past events. For this purpose, we need a possibility, not only to create patterns, but also to query them. In the following example we ask for all events from past where the change in stock price was bigger than 10%.

Example 4.2.

$? - priceChange^{[T_i,T_j]}(X,Y) \otimes X > 10$.

It is also possible to describe negated events. For instance, Example 4.3 represents a $notFulfilledOrder$ event, that triggers when a customer has made a purchase, but the purchase has not been delivered within a certain time. Note that since the event stream is infinite, one should always define a time interval as a scope of a query or a rule. In the Example 4.3 the interval in which we check whether an item has been delivered is $[T_1, T_4]$.

Example 4.3.

$notFulfilledOrder^{[T_1,T_4]}(X) \leftarrow purchased^{[T_1,T_2]}(X) \otimes \neg delivered^{[T_3,T_4]}(X) \otimes (T_4 - T_1 > 3)$.

Example 4.4.

$complexEvent^{[T_1,T_4]}(X) \leftarrow eventA^{[T_1,T_2]}(X) \otimes eventB^{[T_3,T_4]}(X)$

In the following section we give more details about an *execution model* of event patterns specified by Definition IV.

V. Data-driven Complex Event Detection

In Section IV we have specified a complex event pattern. An event pattern is represented declaratively. This section describes how complex events, described with the syntax of $\mathcal{CTR}$, can be effectively detected at run-time.

A. Sequences of Events

We adopt definition for a sequence of events as a set of events ordered by time. For example, the following rules (1) define event patterns based on sequences of events (e.g., e_1 occurs when an event a is followed by an event b, followed by c). Figure 1 depicts a graph representation of events sequences for e_1, e_2 and e_3 (a node with symbol $\otimes$ denotes a sequence). Additionally it shows that e_1 is used to trigger both $action1$ and $action2$, while e_2 and e_3 trigger $action2$ and $action3$, respectively.

Figure 2 gives a graph representation of e_1, e_2 and e_3 too. It is a variant of Figure 1 where each non-atomic event (denoted with $\otimes$) is a two-input node, representing either an intermediate event or a pattern. The latter figure is more convenient to detect a sequence of events, as it compares events two by two (e.g., ie_1 is triggered whenever a is followed by b, and e_1 when ie_1 is followed by c).

$$\begin{array}{l} e_1^{[T1,T6]} \leftarrow a^{[T1,T2]} \otimes b^{[T3,T4]} \otimes c^{[T5,T6]}. \\ e_2^{[T1,T6]} \leftarrow b^{[T1,T2]} \otimes c^{[T3,T4]} \otimes d^{[T5,T6]}. \\ e_3^{[T1,T4]} \leftarrow c^{[T1,T2]} \otimes d^{[T3,T4]}. \end{array} \quad (1)$$

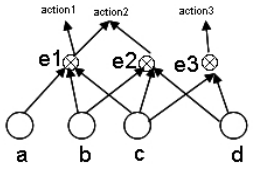

Figure 1. A sequences of events

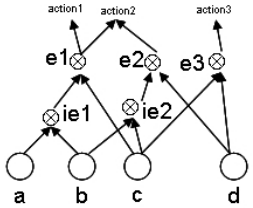

Figure 2. A left-composed sequence of events

Rules (1) specify complex event patterns e_1, e_2 and e_3, created by a user (using the operators from Definition IV). In their present form, these rules are not convenient to be used for *event-driven* computation. They are rather a Prolog-style rules suitable for backward chaining evaluation. Such rules are understood as goals which at certain time either can or cannot be proved by an inference engine. A provable goal means that a corresponding pattern has occurred, and vice versa. Note that backward chaining evaluation is not appropriate for event processing, as it does not detect events when they really occur. Rather it *proves* them at the time when corresponding goals (queries) are set. This is why this kind of event processing is called *query-driven* (Section I).

$$a : -while\_do(a,1). \quad (2)$$

$$a^{[T_1,T_2]}(1) : - ins(goal(b^{[\_,\_]}), a^{[T_1,T_2]}, ie_1^{[\_,\_]})). \quad (3)$$

$$b : - while\_do(b,1). \quad (4)$$

$$\begin{array}{l} b^{[T_3,T_4]}(1) : - goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, ie_1^{[\_,\_]}) \otimes \\ del(goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, ie_1^{[\_,\_]})) \otimes ie_1^{[T_1,T_4]}. \end{array} \quad (5)$$

$$b^{[T_1,T_2]}(2) : - ins(goal(c^{[\_,\_]}, b^{[T_1,T_2]}, ie_2^{[\_,\_]})). \quad (6)$$

$$\begin{array}{l} c : - while\_do(c,1). \\ c^{[T_3,T_4]}(1) : - goal(c^{[T_3,T_4]}, ie_1^{[T_1,T_2]}, e_1^{[\_,\_]}) \otimes \\ del(goal(c^{[T_3,T_4]}, ie_1^{[T_1,T_2]}, e_1^{[\_,\_]})) \otimes e_1^{[T_1,T_4]}. \\ c^{[T_3,T_4]}(2) : - goal(c^{[T_3,T_4]}, b^{[T_1,T_2]}, ie_2^{[\_,\_]}) \otimes \\ del(goal(c^{[T_3,T_4]}, b^{[T_1,T_2]}, ie_2^{[\_,\_]})) \otimes ie_2^{[T_1,T_4]}. \\ c^{[T_1,T_2]}(3) : - ins(goal(d^{[\_,\_]}, c^{[T_1,T_2]}, e_3^{[\_,\_]})). \end{array} \quad (7)$$

$$d :- \ while\_do(d, 1).$$
$$d^{[T_3,T_4]}(1) :- \ goal(c^{[T_3,T_4]}, ie_2^{[T_1,T_2]}, e_2^{[\_,\_]}) \otimes del(goal(c^{[T_3,T_4]}, ie_2^{[T_1,T_2]}, e_2^{[\_,\_]})) \otimes e_2^{[T_1,T_4]}. \quad (8)$$
$$d^{[T_3,T_4]}(2) :- \ goal(d^{[T_3,T_4]}, c^{[T_1,T_2]}, e_3^{[\_,\_]}) \otimes del(goal(d^{[T_3,T_4]}, c^{[T_1,T_2]}, e_3^{[\_,\_]})) \otimes e_3^{[T_1,T_4]}.$$

$$ie_1 : -while\_do(ie_1, 1).$$
$$ie_1^{[T_1,T_2]}(1) : -ins(goal(c^{[\_,\_]}, ie_1^{[T_1,T_2]}, e_1^{[\_,\_]})). \quad (9)$$

$$ie_2 : -while\_do(ie_2, 1).$$
$$ie_2^{[T_1,T_2]}(1) : -ins(goal(c^{[\_,\_]}, ie_2^{[T_1,T_2]}, e_2^{[\_,\_]})). \quad (10)$$

$$while\_do(Pred, N) : -((FullPred = ..[Pred, N]) \otimes execCTR(FullPred) \otimes (N1 is N + 1) \otimes while\_do(Pred, N1)) \vee true. \quad (11)$$

Rules (2)-(10) represent our example rules (1), rewritten so to enable *event-driven backward chaining*. First, they are suitable for detection of events as soon as they occur. Second, detection of events is focused on patterns of interest, rather than merely detecting irrelevant events. These rules (except $''while\_do''$ rules) fall in one of two types of rules. The first type is used to generate goals, and the second one generates either intermediate events (e.g., ie_1, ie_2) or pattern events (e.g., e_1, e_2 or e_3).

In the first type of rules every generated goal represents that an event has occurred. For example, rule (3) inserts $goal(b^{[\_,\_]}, a^{[T_1,T_2]}, ie_1^{[\_,\_]}))$ in the database, every time an event of type a occurs. A goal is a triple, consisting of three events (terms). Its interpretation is that "an instance of event a has occurred at $[T_1, T_2]$, and we are waiting for b to happen in order to detect ie_1", (Figure 2). Obviously the goal does not carry information about times for b and ie_1, as we don't know when they will occur. In general, the second event in a goal always denotes an event that has just occurred. The role of the first event is to specify what we are waiting for to detect an event that is on the third position. In our example, if a was used to build one more pattern (apart from the pattern ie_1), we would have an additional rule which inserts an additional goal. For instance, $goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e_4^{[\_,\_]}))$ could be used to detect e_4 by the same sequence of a and b. In general, the number of rules that insert goals depend on number of *left-to-right* parent edges (see Figure 3(a)).

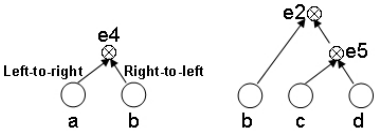

Figure 3. Representation of a goal (a). A right-composed sequence of events (b)

The second type of rules checks whether a certain goal already exists in the database, in which case it triggers an event. For example, rule (5) will fire whenever b occurs. The rule checks whether $goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, ie_1^{[\_,\_]})$ already exists (i.e., a previously has happened), in which case it triggers ie_1 (by calling $ie_1^{[T_1,T_4]}$). The time occurrence of ie_1 (i.e., $[T_1, T_4]$) is defined based on the occurrence of constituting events (i.e., $a^{[T_1,T_2]}$, $b^{[T_3,T_4]}, T_2 < T_3$). Calling $ie_1^{[T_1,T_4]}$, this event is effectively propagated either upward (if it is an intermediate event) or used for triggering an action (if it is event pattern, e.g., e_1).

We see that, in general with backward chaining, goals are crucial for computation of complex events. They show the current state of progress toward matching an event pattern. Moreover they allow for detection of the state of any complex event, at any time. Therefore goals can enable *reasoning* over events, e.g., what are events missing in order to detect some a certain complex event pattern. They can persist over a period of time. It is worth noting that rules of the second type (e.g., rule (5)) also delete goals. Once it is "consumed", the goal is removed from the database[1]. In this way goals are kept persisted as long as (but not longer) they are needed.

Finally rule (11) implements a while-do loop, which for any occurrence of an event goes through each rule specified for that event. For example, when c occurs the first rule in the set of rules (7) will be fired. This rule will then loop, invoking all other rules specified for c (those with c in the rule head), i.e., $c(1)$, $c(2)$ and $c(3)$.

In the following we will briefly explain how rules (2)-(11), for a given events sequence, compute e_1, e_2, e_3 . First, when an instance of a occurs rule (2) will fire and run a loop (i.e., rule (11)). In case of an event a there is only one additional rule (i.e., rule (3)) to be executed in that loop. This rule will insert $goal(b^{[\_,\_]}, a^{[T_1,T_2]}, ie_1^{[\_,\_]})$. Let us assume that some time after, b occurs ($T_2 < T_3$). Similarly this event will fire rule (4), which in turn will fire rules (5) and (6). The first rule will check whether $goal(b^{[\_,\_]}, a^{[T_1,T_2]}, ie_1^{[\_,\_]})$ exists in the database (i.e., a has previously happened), in which case it will trigger $ie_1^{[T_1,T_4]}$. The second rule will insert $goal(c^{[\_,\_]}, b^{[T_1,T_2]}, ie_2^{[\_,\_]})$ denoting that eventual occurrence of c would cause ie_2 to happen. By calling ie_1 from rule (5), this event is effectively propagated upward. A goal (i.e., $goal(c^{[\_,\_]}, ie_1^{[T_1,T_2]}, e_1^{[\_,\_]})$) will be inserted into the database, meaning that an occurrence of c will trigger e_1 (see the second rule in rules (9)). Now when c occurs, rules (7) will all fire. First, it will be checked whether ie_1 has happened, what is required to trigger e_1 (i.e., final complex event). Second, a proof that b has happened will trigger ie_2, which further sets a goal waiting for d. Third, $goal(d^{[\_,\_]}, c^{[T_1,T_2]}, e_3^{[\_,\_]})$ will be asserted. Finally when d occurs, it will fire both complex, events e_2 and e_3.

We have seen how rules (2)-(11) compute complex events in a data-driven backward chaining fashion. Now let us provide more details on structure of these rules and goals. They are designed so to allow a *transformation* from user

[1]In situation where past events are useful for further computation the deletion operation, of course, can be omitted.

defined rules (e.g., rules (1)), and enable efficient event pattern detection. Recall that there exist two types of rules, i.e., one used to generate goals, and another one to trigger events. For any occurrence of an event in the body of user defined rules, one of those two kinds of rules will be added (in the rule base during the transformation). Apart from this, for every particular event (predicate) there will be added one additional rule to define a while-do loop for that event. For instance, there is only one occurrence of a in rules (1), hence there is only one rule added (in a set of rules (2)-(10)) for event a, and one to define a loop. Further on, c occurs three times, thus we have $c(1)$, $c(2)$ and $c(3)$, as well as, one rule for a loop in rules (7). In general, for a given set of rules (defining complex patterns) there will be as many transformed rules as usages of distinguished (atomic) events. The set of transformed rules is further accompanied with rules for loops (as many as there are distinguished atomic events). Finally the same procedure for generating transformed rules is repeated for intermediate events (e.g., ie_1, ie_2). The complete transformation is proportional to the number of user defined event pattern rules, hence such a transformation is linear, and performed at the design time.

In the following, we give more details about computation of intermediate events and goals, which essentially enable the linear transformation. We have already seen that Figure 2 depicts intermediate events by representing them as two-input nodes. An intermediate event is built as a composition of *binary events*. For example, a sequence of events $a \otimes b \otimes c \otimes d$ can be presented as a sequence of binary events $(((a \otimes b) \otimes c) \otimes d)$. This would produce the following intermediate events $ie_1 \leftarrow a \otimes b$, $ie_2 \leftarrow ie_1 \otimes c$ and $ie_3 \leftarrow ie_2 \otimes d$. To effectively handle detection of intermediate events and thus complex patterns, transformed rules (e.g., (2)-(10)) manipulate with insertion and deletion of goals in the database. Note that similarly as an intermediate event, each goal (e.g., $goal(b, a, ie_1)$) constitutes one two-input node, see Figure 2. Two inputs are labeled with a left-hand side event, i.e., an event at the second position in the goal (e.g., a in $goal(b, a, ie_1)$), and a right-hand side event, i.e., an event positioned as the first one (e.g., b). The node itself is labeled with an event at the third position (e.g., ie_1), see Figure 3(a). Two input nodes can be any events, i.e., atomic events, intermediate events or complex events. They are constructed in a direction *left-to-right* (i.e., a *left-composed* events). The reason for this is to avoid *negated goals* (when constructing sequences of events). Figure 3(b) represents a *right-composed* sequence for e_2 (from Figure 2), rules (12)-(15) are created so to implement this sequence. We see that a rule (12), unlike its equivalent rule (6) (from Figure 2), has a negated goal. If this goal did not exist both sequences b, c and d as well as c, b, d would be valid (which is obviously incorrect). In order to avoid unnecessary negated goals we adhere to left-composed events.

$$b^{[T_1,T_2]}(1) :- not(goal(d^{[\_,\_]}, c^{[T_1,T_2]}, ie_2^{[\_,\_]})) \otimes ins(goal(ie_2^{[\_,\_]}, b^{[T_1,T_2]}, e_2^{[\_,\_]})). \quad (12)$$

$$c^{[T_1,T_2]}(1) :- ins(goal(d^{[\_,\_]}, c^{[T_1,T_2]}, ie_2^{[\_,\_]})). \quad (13)$$

$$d^{[T_3,T_4]}(1) :- goal(d^{[\_,\_]}, c^{[T_1,T_2]}, ie_2^{[\_,\_]}) \otimes del(goal(d^{[T_3,T_4]}, c^{[T_1,T_2]}, ie_2^{[\_,\_]})) \otimes ie_2^{[T_1,T_4]}. \quad (14)$$

$$ie_2^{[T_3,T_4]}(1) : -goal(ie_2^{[\_,\_]}, b^{[T_1,T_2]}, e_2^{[\_,\_]}) \otimes del(goal(ie_2^{[\_,\_]}, b^{[T_1,T_2]}, e_2^{[\_,\_]})) \otimes e_2^{[T_1,T_4]}. \quad (15)$$

Finally, we sketch an algorithm that automatically transforms user defined event pattern rules for sequential conjunction into rules which can be used for detection of complex events (using data-driven backward chaining), see Algorithm V-A.

Algorithm V-A Sequential conjunction.

Input: event binary goal $ie_1 \leftarrow a \otimes b$.

Output: event-driven backward chaining rules for $\otimes$ operator.

For each event binary goal $ie_1 : -a \otimes b$. {

whenever a occurs at some $[T_1, T_2]$, apply all rules $r(a)_i$:

$r(a)_1 : a^{[T_1,T_2]} : -ins(goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e1^{[\_,\_]}))$.

whenever b occurs at some $[T_3, T_4]$, apply all rules $r(b)_j$:

$r(b)_1 : b^{[T_3,T_4]} : -goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, ie_1) \otimes T_2 < T_3 \otimes del(goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, ie_1^{[\_,\_]})) \otimes ie_1^{[T_1,T_4]}$. }

Similar simple and clear transformations can be written for the other event operators too. Due to space constraints, in the remaining part of Section V, we briefly describe the other operators for complex events (defined by Definition IV), but leave out the actual transformation algorithms. For a full set of the encoding algorithms of complex events into transaction logic, the full report [2] includes additional transformations and our implementation into XSB Prolog.

B. Other Event Operators

Classical Conjunction of Events. An event pattern based on a classical conjunction occurs when all events which consist that conjunction occur. Unlike the sequence operator, here the constitutive events can happen at different times with no particular order between them. For example, the rule: $e_1^{[T_5,T_6]} \leftarrow e_2^{[T_1,T_2]} \wedge e_3^{[T_3,T_4]}$ is defined as a conjunction of two events, e_2 and e_3. e_1 is detected when both events e_2 and e_3 occur (i.e., either e_2 followed by e_3, or e_3 followed by e_2). An interval in which e_1 is detected (for the given example) is defined by $T_5 = min(T_1, T_3)$ and $T_6 = max(T_2, T_4)$.

Rules (16)-(18) represent rules transformed so to enable detection of the conjunction pattern in an event-driven backward chaining fashion. These rules are created following the same principles (from Subsection V-A) regarding the generation of goals as a mechanism for event detection.

[2]The long technical report and implementation can be found at: http://code.google.com/p/etalis

Therefore we will only briefly describe them, taking into account only differences between them and rules (2)-(11). Due to space limitation, we will also follow this approach in describing the remaining event pattern operators throughout this section.

$$
\begin{array}{l}
e_2 : -while\_do(e_2, 1).\\
e_2^{[T_3,T_4]}(1) : -goal(e_2^{[\_,\_]}, e_3^{[T_1,T_2]}, e_1^{[\_,\_]}) \otimes del(goal(e_2^{[\_,\_]},\\
e_3^{[T_1,T_2]}, e_1^{[\_,\_]})) \otimes min(T_1, T_3, T_5) \otimes max(T_2, T_4, T_6) \otimes e_1^{[T_5,T_6]}.\\
e_2^{[T_3,T_4]}(1) : -not(goal(e_2^{[\_,\_]}, e_3^{[T_1,T_2]}, e_1^{[\_,\_]})) \otimes ins(goal(\\
e_3^{[\_,\_]}, e_2^{[T_3,T_4]}, e_1^{[\_,\_]})).
\end{array} \tag{16}
$$

$$
\begin{array}{l}
e_3 : -while\_do(e_3, 1).\\
e_3^{[T_3,T_4]}(1) : -goal(e_3^{[\_,\_]}, e_2^{[T_1,T_2]}, e_1^{[\_,\_]}) \otimes del(goal(e_3^{[\_,\_]},\\
e_2^{[T_1,T_2]}, e_1^{[\_,\_]})) \otimes min(T_1, T_3, T_5) max(T_2, T_4, T_6) \otimes e_1^{[T_5,T_6]}.\\
e_3^{[T_3,T_4]}(1) : -not(goal(e_3^{[\_,\_]}, e_2^{[T_1,T_2]}, e_1^{[\_,\_]})) \otimes\\
ins(goal(e_2^{[\_,\_]}, e_3^{[T_3,T_4]}, e_1^{[\_,\_]})).
\end{array} \tag{17}
$$

$$
\begin{array}{l}
min(T_1, T_2, T_3) : -(T_1 < T_2 \rightarrow T_3 = T_1; T_3 = T_2).\\
max(T_1, T_2, T_3) : -(T_1 > T_2 \rightarrow T_3 = T_1; T_3 = T_2).
\end{array} \tag{18}
$$

Let us assume that e_2 was detected as the first event (followed by e_3 afterwards). In this situation, rules (16) will fire. The loop will execute two rules. The first rule will check whether e_3 has previously happened, in which case it will trigger $e_1([T_5, T_6])$, provided that $T_5 = min(T_1, T_3)$ and $T_5 = max(T_2, T_4)$. However if e_3 had not previously happened (by the time when e_2 occurred), the second rule will insert $goal(e_3^{[\_,\_]}, e_2^{[T_3,T_4]}, e_1^{[\_,\_]})$. This goal has the meaning that e_2 occurred at $[T_3, T_4]$, and now we are waiting for e_3 to happen in order to detect e_1. Once e_3 occurs, rules (17) will execute. Again there is a loop firing two rules. The first rule will trigger e_1 (provided that e_2 has already happened). A time interval on which e_1 has been detected is the same, i.e., $T_5 = min(T_1, T_3)$ and $T_6 = max(T_2, T_4)$. If e_2 has not happened, a goal stating that e_3 happened (in scope of detection of e_1) is inserted. It is worth noting that a non-occurrence of an event is checked by using a negated goal. In Subsection V-A we have discussed left and right-composed nodes in order to avoid use of negated goals. As we see, for the classical conjunction operator we still need negated goals. However their use for this operator is not an issue. In case of right-composed nodes of a sequence of events, an algorithm for the rule transformation would need to add one negated goal to an existing rule every time it encounters a new node (in that sequence). That is, the algorithm would need to *continuously modify* already created transformation rules as long as the transformation takes place. As shown, this can be avoided by using left-composed nodes in a sequence of events.

Concurrent Conjunction of Events. A concurrent conjunction of two events is detected when their intervals overlap. For example, let us consider a rule representing a concurrent conjunction: $e_1 \leftarrow e_2 | e_3$. e_1 is detected if events e_2 and e_3 both occur, and their detection intervals overlap.

As classical and concurrent conjunction are similar operators, to detect e_1 we can still use rules (16)-(18). However we need to introduce a condition that T_3 must strictly be less than T_2 (i.e., $T_3 < T_2$). At the same time, built-in predicates for finding min and max times in both rules become unnecessary (hence they can be removed). e_1 will be defined on an interval $[T_1, T_4]$ (instead of $[T_5, T_6]$). The *less* condition ensures that intervals of e_2 and e_3 overlap. Note that, due to that condition, a concurrent conjunction of events can be built only with non-atomic events.

Disjunction of Events. Disjunction is detected when any of events that constitutes a disjunction occurs. For instance, a rule: $e_1 \leftarrow e_2 \vee e_3$ defines e_1 as a disjunction of two complex events, e_2 and e_3, and is transformed into as many pattern rules as many disjuncts exist (i.e., two in our example). If disjuncts consist of complex patterns, they are further split in other event operators and processed.

Negation. Negation in event processing is typically understood as absence of that event. In order to create a time interval in which we are interested to detect absence of an event, we define a negated event in scope of other complex events. Let us use a sequence to form a time interval in our example. A rule (19) defines a pattern for e_1 detected whenever an event a is followed by b, provided that c does not happen in between.

$$e_1^{[T_1,T_4]} \leftarrow (a^{[T_1,T_2]} \otimes b^{[T_3,T_4]}) \wedge \neg c^{[T_5,T_6]}. \tag{19}$$

$$a^{[T_1,T_2]}(1) : -ins(goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e_1^{[\_,\_]})). \tag{20}$$

$$
\begin{array}{l}
b^{[T_3,T_4]}(1) : - goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e_1^{[\_,\_]}) \otimes\\
del(goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e_1^{[\_,\_]})) \otimes e_1^{[T_1,T_4]}.
\end{array} \tag{21}
$$

$$c^{[T_1,T_2]}(1) : - del(goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e_1^{[\_,\_]})). \tag{22}$$

Rules (20)-(22) sketch how negation can be handled. Similarly as in Subsection V-A, a sequence of a and b is detected by rules (20)-(21). The difference is a rule (22). This rule is created to delete a goal that was inserted by a (if such a goal exists, hence it needs to be checked first). Whenever c happens after a, it will reset the current state of detection of e_1. In turn, e_1 is detected after an occurrence of b, which follows a, with no c in between.

VI. Experimental Results

We have implemented algorithms that transform user defined event pattern rules into rules suitable for event-driven pattern detection. The test cases (that we show here) were run on a workstation with Pentium dual-core processor 2,4GHz CPU and 3GB memory running on Ubuntu Linux and XSB Prolog version 3.1. In our experiments we have used different operators to generate a few events in a *block* of complex patterns.

The first experiment consists of six atomic events (i.e., e_1-e_6) and four complex events (i.e., ce_1 - ce_4), see experiment rules (23) below. Further, we have been multiplying the number of *event blocks* to assess the throughput of input events that can be handled by our prototype.

$$
\begin{array}{l}
ce_1 : -e_1 \otimes e_2. \\
ce_2 : -(e_2 \otimes e_3) \wedge \neg e_4. \\
ce_3 : -(e_3 \otimes e_5) | (e_4 \otimes e_6). \\
ce_4 : -e_2 \vee (e_3 \wedge ce_2).
\end{array} \tag{23}
$$

Figure 4 shows the achieved experimental results. The test is repeated an increasing number of times. The maximum throughput that we got is in a range between 12600 and 13500 events/second. We tested sizes of 10,000 blocks (4438ms), 50,000 (21360ms) and 500,000 (237950ms).

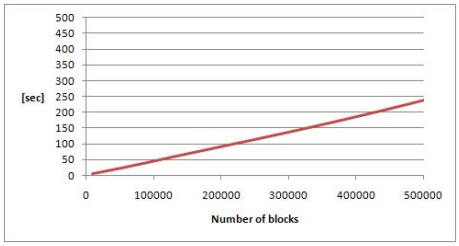

Figure 4. The event throughput for our experiments

The memory consumption during the pattern matching process is very low and constant. Irrelevant events are removed from memory as soon as they are "consumed".

The second experiment consists of compiling the real end-of-day market stock values as event triggers for the IBM company (in the last 40 years, approx. 10K stock ticks). Each tuple has: the date, the opening and closing price, and the low and the high for the day. A composed event is triggered when the opening price is higher then the closing price, see rule (24). The execution time was 9.408sec, which is rather worse performance (approx. 1000 events/second) in comparison to the first experiment. The reason is twofold. First, in the latter experiment we deal with real stock events (i.e., with a set of data carried with each events). This data is indexed on the first attribute in XSB, which is inefficient for our experiment. Second, as real stock events are represented as predicates (not as propositions, see rules (23)), their processing is more complex.

$$
\begin{array}{l}
ce3(Date) : -stock('IBM', Volume, Date, \\
Open, Close, Low, High) \otimes Open > Close.
\end{array} \tag{24}
$$

In this section we have provided the first measurement results. Even though there is a lot of room for improvements and optimizations, preliminary results show that logic-based event processing has capability to handle significant amount of events in reasonable time. Taking account its strength (i.e., robust inference capability), it promises a powerful approach for combining *deductive* and *temporal* capabilities in a unified framework, yet with good run-time characteristics.

VII. Conclusions and Future Work

We propose a novel approach for Complex Event Processing based on an encoding of event patterns into logic programs. Particularly our approach is inspired by Transaction Logic. The approach clearly extends capabilities of Active Databases with declarative semantics and the power of rule-based reasoning. We have presented a transformation which converts user defined complex event patterns into, so called, event-driven backward chaining rules. These rules allow for data-driven computation of complex events, and reasoning about them. We have devised an algorithm which does this conversion automatically. Further on, we have provided a prototype implementation, and first evaluation results of our approach. For the next steps we will continue to study advantages and drawbacks of Complex Event Processing implemented in a logical framework, and work on the implementation of transformation algorithms for new event operations (e.g., the isolation operator, time window operators, aggregates etc.).

Acknowledgments

The authors would like to thank Ahmed Khalil Hafsi and Jia Ding for their help in testing of our prototype.

References

[1] F. Bry and M. Eckert, "Rule-based composite event queries: The language xchangeeq and its semantics," in *RR*. Springer, 2007.

[2] A. Paschke, A. Kozlenkov, and H. Boley, "A homogenous reaction rules language for complex event processing," in *International Workshop on Event Drive Architecture for Complex Event Process*. ACM, 2007.

[3] P. Haley, "Data-driven backward chaining," in *International Joint Conferences on Artificial Intelligence*. Milan, Italy, 1987.

[4] E. Behrends, O. Fritzen, W. May, and F. Schenk, "Combining eca rules with process algebras for the semantic web," in *RuleML*, 2006.

[5] N. W. Paton and O. Díaz, "Active database systems," in *ACM Comput. Surv.* ACM, 1999.

[6] R. Milner, "Calculi for synchrony and asynchrony," in *Theor. Comput. Sci.*, 1983.

[7] A. J. Bonner and M. Kifer, "Concurrency and communication in transaction logic," in *Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996.

[8] ——, "Transaction logic programming (or, a logic of procedural and declarative knowledge," in *Technical Report CSRI-270*, 1995.

[9] D. Roman and M. Kifer, "Reasoning about the behavior of semantic web services with concurrent transaction logic," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowmentg, 2007.